UTILIB User Manual Reference Manual

Version 3.0

William E. Hart

September 22, 2006

Abstract

This document describes the UTILIB software library. UTILIB includes a variety of generic components for C++ software development including abstract data types, I/O management, sorting routines, and random number generators. The UTILIB library is a core component of the Acro optimization framework, and it has been used separately for other projects at Sandia, including DAKOTA and NETV.

Contents

1 Introduction **1**

2 Abstract Data Types **3**

3 Mathematical Routines **13**

4 Random Numbers and Random Variables **14**

5 Class Parameter Initialization **16**

6 Parallel Computing Utilities **20**

7 Miscellaneous Classes and Utilities **26**

8 Managing Exceptions **28**

9 Acknowledgements **30**

1 Introduction

1.1 Overview

UTILIB is a general-purpose C++ library that includes a variety of algorithmic utilities for software development. These utilities define useful datatypes and classes as well as generic routines. In particular, UTILIB provides a variety of services that facilitate the portability of codes, and in particular porting to parallel computing platforms at Sandia. This library has proven useful in the development of several codes at Sandia, including the Coliny optimization library, the PICO parallel branch-and-bound library, and the DAKOTA optimization toolkit.

It is worth noting some points about the design philosophy for the classes in UTILIB:

- **Encapsulation:** One of the chief advantages for using UTILIB data types (e.g. arrays) is the encapsulation of memory allocation that they provide. This feature has been heavily exploited in my subsequent code, and thus memory allocation is generally quite robust. Further, some classes (e.g. `LinkedList`) include mechanisms for efficiently 'reallocating' memory.

- **Robustness:** A related aspect of UTILIB's design is robustness. I have almost always favored design considerations that ensure robustness. For example, the default behavior for `BasicArray's` is to perform bounds checking. In practice, the performance hit that this causes has been far outweighed by the hours saved tracking down obscure errors.

- **Portability:** Portability across many different architectures is another very important aspect of UTILIB. For example, the common definitions for sorting in **sort.h** have proven very effective for defining portable sorting routines.

- **Efficiency:** There is generally no *best* way to implement many algorithms and datatypes, since there invariably are performance/utility trade-offs that need to be made. In the design of UTILIB classes, I have generally looked for solutions that admit a reasonably efficient capability while providing the most general possible design. For example, ADT's like splay trees are very general in their capabilities. Still, they include methods that allow the user to track pointers to items in the tree, which can later be used to efficiently remove those items from the tree.

- **Parallelization:** Support for parallelization is an important function for UTILIB. In particular, UTILIB includes mechanisms for managing parallel I/O through the `CommonIO` class, and the `uMPI` class provides wrappers for parallel communication with MPI.

This user guide is focused on describing the capabilities and software components in the UTILIB library. The components of the UTILIB library include

- **Abstract Data Types:** standard abstract data types like trees and arrays

- **Input/Output Routines:** facilities for encapsulating error routines as well as redirecting I/O through user-defined streams

- **Mathematical Routines:** commonly used mathematical routines, especially array operations

- **Random Number Generation:** generators for commonly used probability distributions and a portable random number generator

- **System Support:** miscellaneous routines, especially to support portability between different operating systems

These components of the libraries are described in greater detail in the following sections. For further details on UTILIB, including instructions for downloading and installing this software, see the Acro web pages: `http://software.sandia.gov/Acro`.

1.2 Changes in UTILIB 3.0

Although an official 2.0 of UTILIB was not distributed, there have been significant changes since the 1.0 release several years ago. In particular, there recently have been very significant changes to UTILIB, and from a user's perspective UTILIB looks very different from the way it did six months ago. The 3.0 release signifies these changes, which are summarized as follows:

- A complete rework of the configuration management process to use autoconf tools.
- Integration of UTILIB with the Acro framework, which supports nightly testing on a wide range of computing platforms (including native Windows builds with MINGW).
- Rework of many abstract data types to use iterator mechanisms.
- Elimination of the sorting codes, but explicitly leveraging of STL sorting to support a variety of sorting-related activities.
- The extension of this document to include all aspects of UTILIB.

2 Abstract Data Types

UTILIB includes classes for a variety of standard abstract data types:

- Arrays and Matrices
- Hash Tables
- Heaps
- Linked Lists, Queues and Stacks
- Sets
- Splay Trees

A detailed description of these data types is beyond the scope of this user guide, but these data structures are widely described by a variety of introductory algorithms texts [1, 3, 4, 7, 8].

2.1 Arrays and Matrices

UTILIB defines several types of array classes: simple arrays, dense matrices (two-dimensional arrays), sparse matrices, three-dimensional arrays, enumeration arrays, and bit arrays.

2.1.1 Simple Arrays

The **utilib::BasicArray** class provides a nice level of encapsulation for array data types, and the **utilib::NumArray** class extends this class to include numerical vector operations. The primary advantage of using these classes over similar classes in STL is that UTILIB includes extra safety features such as runtime bounds checking and reference counting.

These array classes have a pointer to the data in the field `Data`. You can get this pointer using the data() function:

```
utilib::BasicArray<int> vec1(5);
vec1 << 0;
int *intarray = vec1.data();
```

This example uses a constructor that specifies a vector of length five. You can also specify the initial content of a **utilib::BasicArray** explicitly:

```
int array_of_ints[20];
utilib::BasicArray<int> vec2(20, array_of_ints);
```

The first argument is the length of the array and the second is a pointer to an array of the appropriate type. The data field in the **utilib::BasicArray** will point to this array. One can also construct a copy of an existing array:

```
utilib::BasicArray vec3(vec2);
```

This will allocate a new integer vector and initialize it with the contents of `vec2`. The Data field in `vec3` points to the new copy.

Frequently, one will need to use an empty constructor (i.e. start with a size-zero array) and then put in the true data. More generally, you may want to grow and shrink the vector dynamically:

```
vec3.resize(100);
```

This will re-allocate a new array of 100 integers and copy the old data (if any) into the beginning of the array. In this example, the first twenty elements are copied from the previously allocated memory for `vec3`. For a **utilib::BasicArray** object you cannot assume anything about the remaining values, but for NumArray objects these array elements will be initialized to zero. If an array is resized to a smaller value, then the first part of the previous array is retained.

```
vec3.size()
```

returns the length of `vec3`.

The equals (=) operator allocates new space. Thus

```
utilib::BasicArray vec4;
vec4 = vec3;
```

creates a new integer array and copies the contents of `vec3` into that array. The Data field of `vec4` points to the new space. If the vector already exists and you want to reuse the already-allocated space, use the « operator:

```
utilib::BasicArray vec5(100);
vec5 << vec4;
```

This copies the contents of the Data array from `vec4` into the array for `vec5`. If the allocated array (`vec5`) is not the same size, then it will be resized by this operator. To copy by reference, use the `&=` operator. Thus

```
utilib::BasicArray<int> vec6 &= vec5;
```

will have the data of `vec6` point to the same array that the data of `vec5` points to (reference counts are properly updated).

The stream (<<) operator is overloaded to allow (re)initialization of a vector that has already been created.

```
vec5 << 15;
```

This sets every element of `vec5` to 15.

Getting array elements works looks like normal array references:

```
int index, newvalue;
vec5[index] = newvalue;
newvalue = vec5[index];
```

Iterators are provided for **utilib::BasicArray** and **utilib::NumArray**, which have the same look-and-feel as those used for the STL `std::vector` class. For example:

```
utilib::BasicArray<int>::iterator curr = vec5.begin();
utilib::BasicArray<int>::iterator end  = vec5.end();
while (curr != end) {
  cout << *curr << " ";
  curr++;
}
cout << *curr << endl;            /// ERROR HERE!
```

However, these iterators also detect whether the iterator has gone beyond the bounds of the array. In the previous example, the iterator would throw an exception at the last step, when the value of `curr` is being referenced.

Notes:

- The **utilib::pvector** class extends the STL `std::vector` class to include some of the bounds checking that is used by **utilib::BasicArray**.
- The **utilib::IntVector** class is an alias for **utilib::NumArray**<**int**>, and the **utilib::DoubleVector** class is an alias for **utilib::NumArray**<**double**>
- The **utilib::MixedIntVars** class is a simple container for one-dimensional arrays of doubles, integers and binary values.
- Error checks for bounds an iterators cannot be turned off right now.

2.1.2 Character String

The **utilib::CharString** class is a subclass of **utilib::BasicArray** that provides additional functionality for manipulating character strings. This class has a similar look-and-feel to the STL `std::string` class, though **utilib::CharString** is more heavily integrated into various UTILIB utilities.

A variety of comparison operations are supported for strings (using the same lexicographical ordering as `strcmp()`). For example,

```
utilib::CharString str_a = "abc";
utilib::CharString str_b = "xyz";
cout << (str_a == str_b); /// False
cout << (str_a != str_b); /// True
cout << (str_a < str_b); /// True
cout << (str_a >= str_b); /// False
```

Similarly, this class include methods for easily setting strings:

```
utilib::CharString str;
str = "file_";
str += 0;
cout << (str == "file_0"); /// True;
```

Otherwise, this class looks much like a **utilib::BasicArray**. One additional difference is that the `c_str()` method can be used to extract the underlying character string (as is done for `std::string`).

2.1.3 Matrices

Dense and sparse matrices are supported by UTILIB. The **utilib::Basic2DArray** and **utilib::Num2DArray** classes define dense matrices as an array-of-arrays. Similarly, three-dimensional dense matrices are defined in **utilib::Basic3DArray**. NOTE: These classes are not widely used, and thus they are not as mature as the one-dimensional array classes.

The **utilib::SparseMatrix** class is a base class for defining sparse matrices. **utilib::CMSparseMatrix** defines sparse matrices with column-major ordering, and **utilib::RMSparseMatrix** defines sparse matrices with row-major ordering. These classes are heavily used by the PICO MILP solver, so they are relatively stable. Sparse matrices can be setup by adjoining column or rows, or by initializing the sparse data structures directly. To do the later, you need to initialize the following data (using a column-major ordering for example):

- matval[i] - The array of values in the sparse matrix.
- matind[i] - The row-indeces of the corresponding array values.
- matbeg[i] - The index of `matval` that is the beginning of the i-th column.
- matcnt[i] - The length of the i-th column.

Notes:

- The **utilib::IntMatrix** class is an alias for utilib::Num2DArray<int>, and the **utilib::DoubleMatrix** class is an alias for **utilib::Num2DArray**<**double**>.

2.1.4 Enumerated and BitArrays

The **utilib::BitArray** class employs a compressed data representation for boolean data. The main elements of this array have the same look and feel as a **utilib::BasicArray** object. The elements of this array can be changed using the `set()` and `reset()` methods, which alternatively turn the array values to `true` and `false`. For example:

```
utilib::BitArray array(5);
array.set(); /// Turn all values on
array.reset(3);         /// Turn off the 4th value
array.set(3);                       /// Turn the 4th value back on
array.put(2,0); /// Turn off the 3rd value
```

The **utilib::BitArray** object can also use standard array notation:

```
utilib::BitArray array(5);
array[0] = false; /// Turn off the 1st value
array[1] = true; /// Turn on the 2nd value
```

The **utilib::TwoBitArray** class provides a similar compressed array representation for arrays with values 0, 1, 2 or 3. More generally, the **utilib::EnumBitArray** can be used to represent arrays for enumeration types that can be coerced to integer values.

2.2 Linked Lists, Queues and Stacks

The **utilib::LinkedList** class defines a doubly-linked list. The look-and-feel of **utilib::LinkedList** is quite similar to the STL `std::list` class. Thus, general documentation of `std::list` is generally accurate for this class (e.g. see `http://www.sgi.com/tech/stl/List.html`). Doubly linked lists support O(1) insertion and deletion, but searching these data structures is O(n). For example, the following example illustrates the setup of a simple list:

```
utilib::LinkedList<int> mylist;
//
// Initializing a list with integers 0..9
//
for (int i=0; i<10; i++)
  mylist.push_back(i);
//
// Printing the list
//
utilib::LinkedList<int>::iterator curr = mylist.begin();
utilib::LinkedList<int>::iterator end  = mylist.end();
while (curr != end) {
  cout << *curr << " ";
  curr++;
  }
```

The **utilib::LinkedList** object supports some additional features that differentiate it from the STL std::list class:

- The `add()` and `remove()` methods can be used to insert and delete elements from a list such that the list acts like a queue (inserting and deleting from the front) or a stack (inserting and deleting from the end). This behavior can be configure with the `stack_mode()` and `queue_mode()` methods. The **utilib::QueueList** and **utilib::StackList** classes leverage these mechanisms explicitly.

- The **utilib::LinkedList** iterators validate that an iterator is valid before they are referenced to retrieve the value. In the case of an error, an exception is thrown that can be used to trap the error.

- The **utilib::LinkedList** manages a list of **utilib::ListItem** objects that store the data in the list. The **utilib::CachedAllocator** class is used to cache these objects when they are deleted. This minimizes memory allocations when lists are repeatedly filled and cleared.

The **utilib::OrderedList** class is a derived linked list object that orders items that are inserted into the list. This is an O(n) operation, so this is generally not a useful class. However, this provides a simple ordered container for short ordered lists.

2.3 Heaps

The class **utilib::AbstractHeap** defines an abstract class that provides the core operations of a heap. This is adapted from code developed by Jonathan Eckstein (Rutgers). A heap is a partially sorted binary tree. The heap's tree is not completely in order, but it ensures that every node has a value less than either of its children. Additionally, a heap is a "complete tree" – every level of the tree is filled in before adding a node to the next level, and one that has the nodes in a given level filled in from left to right, with no breaks. Items can be inserted into and removed from heap with O(log n) effort.

The **utilib::SimpleHeap** class is a simple heap object that maintains copies of the keys that are kept in the tree. The **utilib::GenericHeap** class maintains references to the keys that are kept in the tree. The `add()` and `remove()` methods are used to insert and delete items this tree. Iterators are not currently supported for heaps, but references to elements of this tree are returned by the `add()` and `remove()` methods.

The following example illustrates the use of a heap:

```
utilib::SimpleHeap<int> tree;
//
// Initializing a heap with integers
//
for (int i=0; i<10; i++)
  tree.add(200*i % 13);
//
// Printing and deleting the tree (in sorted order)
//
for (int i=0; i<10; i++) {
  SimpleHeapItem<int>* item = tree.top();
  cout << item->key() << " ";
  bool status;
  tree.remove(item, status);
  }
```

2.4 Hash Tables

The class **utilib::AbstractHashTable** defines an abstract class that provides the core operations of a hash table with chaining. Hash tables are data structures that are used when you are managing a large amount of data and need to be able find an item quickly. A hash table uses a hash function that transforms the key to an integer that provides an index into a table (or array). In general, it is impossible to prevent collisions, where two different keys are hashed to the same index. To counteract this, this hash table dynamically resizes the array to ensure that hashed keys are sparsely represented. Further, conflicts are resolved by chaining, which uses a linked list of elements at a given point in the hash table.

The **utilib::SimpleHashTable** class is a simple hash table object that maintains copies of the keys that are kept in the table. The **utilib::GenericHashTable** class maintains references to the keys that are kept in the table. The `add()` and `remove()` methods are used to insert and delete items this tree. Iterators are supported for hash tables, though the data is in an arbitrary order within the iterator.

The following example illustrates the use of a hash table:

```
utilib::SimpleHashTable<int,char> ht;
//
// Initializing a heap with integers
//
char foo;
for (int i=0; i<10; i++)
  ht.add(200*i % 13, foo);
//
// Printing a hash table (in an arbitrary order)
//
utilib::SimpleHashTable::iterator curr = ht.begin();
utilib::SimpleHashTable::iterator end  = ht.end();
while (curr != end) {
  cout << *curr << " ";
  curr++;
  }
```

The hash functions used for these hash tables are defined in **hash_fn.h**. Many of these functions are based on the Bob Jenkins hash function, which are discussed in detail at `http://burtleburtle.net/bob/hash` . In particular, these hash functions do not require that the hash table have a prime length.

The **utilib::LPHashTable** class defines a limited precision hash table (for arrays of doubles). The **utilib::LPHash-Table** class is derived from **utilib::AbstractHashTable**, which defines the basic operations of the hash table. The keys are assumed to be classes for which the following operations are defined:

- `size_type hash( size_type tablesize, unsigned int precision)`
- `const int compare( KEY& key)`
- `const int write( ostream& os)`
- `const int read( istream& is)`

2.5 A Hashed Set Class

A container class that stores a set of values such that each value is represented uniquely. For example, the following example illustrates the setup of a simple list:

```
utilib::HashedSet<int> mylist;
//
// Initializing a list with integers 0..9
//
for (int i=0; i<10; i++)
  mylist.insert(i);
//
// Printing the list
//
utilib::HashedSet<int>::iterator curr = mylist.begin();
utilib::HashedSet<int>::iterator end  = mylist.end();
while (curr != end) {
  cout << *curr << " ";
  curr++;
  }
```

This class is akin to the STL `std::set` class, though they have different APIs, and a hashtable is used to manage the underlying data structure. Thus, the set is not stored in order, and insertions can be performed with O(1) effort.

2.6 Splay Trees

The class **utilib::AbstractSplayTree** defines an abstract class that provides the core operations of a top-down splay tree. This is adapted from code developed by D. Sleator, which itself is adapted from simple top-down splay, at the bottom of 669 of Sleator and Tarjan [6].

Splay trees are a simple and efficient data structure for storing an ordered set. The data structure consists of a binary tree, with no additional fields. It allows searching, insertion, deletion, deletemin, deletemax, splitting, joining, and many other operations, all with amortized logarithmic performance. Since the trees adapt to the sequence of requests, their performance on real access patterns is typically even better.

The splay operation is applied to a binary search tree. It restructures the tree as it descends toward the desired key's place in the tree. During descent, long paths are shortened by rotation. Ultimately, when the desired key is found, the binary search tree is reassembled to make the desired key's node the new root.

The **utilib::SimpleSplayTree** class is a simple splay tree object that maintains copies of the keys that are kept in the tree. The **utilib::GenericSplayTree** class maintains references to the keys that are kept in the tree. The `add()` and `remove()` methods are used to insert and delete items this tree. Iterators are not currently supported for splay trees, but references to elements of this tree are returned by these operators.

The following example illustrates the use of a splay tree:

```
utilib::SimpleSplayTree<int> tree;
//
// Initializing a tree with integers
//
for (int i=0; i<10; i++)
  tree.add(200*i % 13);
//
// Printing the tree
//
for (int i=0; i<10; i++) {
  SimpleSplayTreeItem<int>* item = tree.find_rank(i);
  cout << item->key() << " ";
  }
```

3 Mathematical Routines

The **_math.h** header includes several headers that define mathematical and array functions:

- **math_basic.h** - Defines basic mathematical routines and constants.
- **math_array.h** - Defines mathematical routines that are applied to arrays.
- **math_matrix.h** - Defines mathematical routines that are applied to matrices.

Note that the **linpack.h** header is not currently used in UTILIB.

3.1 Comparison Mechanisms

Many routines in UTILIB perform a comparison between two objects and return an integer flag. If we are evaluating how A relates to B, then the standard comparison semantics for the return value x is that x is less than zero if A is before B in the order, x is greater than zero if A is after B in the order, and x is zero if they are equal. In the context of numerical values, A is before B if A is less than B. Finally, note that if the comparison function is a method of an object, like

```
A.compare(B)
```

then the comparison is evaluating how A relates to B (and not how B relates to A).

The **utilib::ComparisonBase** class defines a generic mechanism for defining comparison class. Two subclasses of this have been developed:

- **utilib::SimpleCompare** - A simple comparison class
- **utilib::Reverse** - A class for performing a reverse-ordered comparison

3.2 Sorting

The **sort.h** header contains definitions for a variety of comparison, sorting and ordering functions:

- sort - Sort an array object.
- stable_sort - Perform a stable sort on an array object.
- order - Fill an array ndx with the order of the elements. Thus, ndx[i] is the index of the i-th smallest value in the array.
- rank - Fill an array ndx with the rank of the elements. Thus, ndx[i] is the rank of the i-th element of the array.

These functions employ the STL sorting routines, which are portable and robust. Further, there are various instances of these functions that are applicable to **utilib::BasicArray** and std::vector objects.

4 Random Numbers and Random Variables

UTILIB supports a variety of classes for defining and using random generators as well as classes for generating samples from different types of random variables.

4.1 Random Number Generators

The basic datatype for random number generators is **utilib::RNG**, and two classes are provided for encapsulating linear congruential generators:

- **utilib::LCG** - encapsulates the unix random number generator, and
- **utilib::PM_LCG** - encapsulates a portable random number generator [5].

The **default_rng.h** header provides an API for initializing and using a global **utilib::PM_LCG** random number generator (which is particularly convenient for C code).

The **utilib::AnyRNG** class is used to maintain a reference to *some* **utilib::RNG** object that has been created (or, in fact, any object that uses the same API as **utilib::RNG**). This works like the **utilib::AnyReference** object, in that it can store a reference to any such object. This utility was developed to enable codes to be developed with a generic container class while allowing users to provide their favorite random number generator object. This object provides more error checking than, say, a pointer to a **utilib::RNG**.

4.2 Random Variables

UTILIB includes a variety of classes for generating random variables using a random number generator. In all cases, a random variable object `rv` acts like a functor when generating a random value:

- **utilib::Binomial** - `rv(p,n)` generates a value from the binomial distribution with probability `p` and number of trials `n`.
- **utilib::Cauchy** - `rv(alpha,beta)` generates a value from the Cauchy distribution with parameters `alpha` and `beta`.
- **utilib::DUniform** - `rv(l,h)` generates a uniformly random integer value from `l` to `h`.
- **utilib::Exponential** - `rv(m)` generates a value from the exponential distribution with mean `m`.
- **utilib::Geometric** - `rv(p)` generates a value from the geometric distribution with probability of success `p`.
- **utilib::LogNormal** - `rv(scale,shape)` generates a value from the log-normal distribution with parameters `scale` and `shape`.
- **utilib::Normal** - `rv(m,s)` generates a value from the normal (or Gaussian) distribution with mean `m` and standard deviation `s`.
- **utilib::Uniform** - `rv(l,h)` generates a value from the uniform distribution between low `l` and high `h`.

- **utilib::Triangular** - `rv(l,h)` generates a value from the triangular distribution between low `l` and high `h`.

Additionally, the following

- **utilib::MNormal** - rv(vec) generates a vector of random numbers from a normal distribution, using a user-specified covariance matrix.

- **utilib::MUniform** - rv(vec) generates a vector of random numbers from a uniform distribution, where the range may vary for eaCh dimension.

These random variables distributions are described in detail in a variety of texts (e.g. Evans, Hastings and Peacock [2]). Many of these classes are implemented using code from the RANLIB random number generator library (see the **Random.h** header file).

The following base classes are used to define random variable objects:

- **utilib::RandomVariableBase** - Abstract class for random variables.

- **utilib::SimpleRandomVariable** - An abstract templated random variable class that generates values that are returned by value.

- **utilib::GeneralRandomVariable** - An abstract templated random variable class that generates values that are returned by reference.

- **utilib::ExternalRandomVariable** - A **utilib::SimpleRandomVariable** that generates points using an external function.

4.3 Sample Generators

The **utilib::SampleGenerator** class is an abstract base class for objects that generate a set of points sequentially. Although the random variable classes can be used in this fashion, there are often contexts where the sample is a function of all of the points that are sampled. Thus, a **utilib::SampleGenerator** subclass can contain context about the previously generated points for use when generating new points.

The **utilib::UniformSampleGenerator** template class is a simple implementation where each sample is generated independently. This class is specialized for generating **utilib::BasicArray**<**double**> and **utilib::MixedIntVars** samples.

5 Class Parameter Initialization

UTILIB contains definitions of several classes that can be used to control the initialization of name-value pairs that can be used to initialize classes:

- **utilib::Parameter**
- **utilib::ParameterSet**
- **utilib::ParameterList**

The **utilib::Parameter** class provides a container object that can provide a reference to another data type. As such, a **utilib::Parameter** instance provides mechanisms for initializing and retrieving the value of this data.

The **utilib::ParameterSet** class manages sets of **utilib::Parameter** objects, and as such this class will probably be used more directly than the **utilib::Parameter** class. Since **utilib::Parameter** objects provide a reference to data, **utilib::ParameterSet** objects can be used to initialize data in a class in a transparent manner. For example, consider a class

```
class Example1
{
  Example1()
      { a=1; }

  void print()
      { cout << a << endl; }

  int a;
};
```

We can extend this class to support parameter initialization by (1) making Example1 a subclass of **utilib::ParameterSet** and (2) creating a parameter in the constructor.

```
class Example1 : public utilib::ParameterSet
{
  Example1()
      {
      a=1;
      create_parameter("a",a,"<int>","1","Class data for Example1");
      }

  void print()
      { cout << a << endl; }

  int a;
};
```

The **utilib::ParameterSet::create_parameter()** method creates a **utilib::Parameter** object in the **utilib::ParameterSet** base class of Example1. This parameter object can be used to initialize the data in an Example1 object as illustrated below:

```
Example1 obj;                           // Create an Example1 object
obj.print();                            // Prints "1"
obj.set_parameter<int>("a",10);         // Set the value of obj.a to 10
obj.print();                            // Prints "10"
```

Note that this use of a parameter class is qualitatively different from a design that stores parameters by value (e.g. as used by Sandia's Trilinos package). If parameters are stored by value, then we cannot transparently initialize classes as we have illustrated here. However, a database of parameter values can be managed independent of any class instantiation. This can, for example, allow parameters to be passed around and referenced where needed in user code.

The following sections provide more detail for how UTILIB parameters can be used in practice.

5.0.1 Common ParameterSet Methods

The most basic operations for a **utilib::ParameterSet** object are the get/set methods, which are illustrated below:

```
Example1 obj;

obj.set_parameter("a",1);                  // Set a parameter with a value

obj.set_parameter_with_string("a","1");    // Set a parameter with a string
                                           //  which is interpreted as a
                                           //  value with the parameter's type

int value;
obj.get_parameter("a",value);              // Retrieve a parameter value using
                                           // a given data element

value = obj.get_parameter<int>("a");       // Return a parameter value using
                                           // a specified return type
```

For all of these **utilib::ParameterSet** methods, checks are made to ensure consistency of the underlying parameter type with the types specified by the user. If these checks fail, an exception is generated that throws std::runtime_error.

In addition to these methods, parameters can be initialized in a **utilib::ParameterSet** object with command-line arguments and file input. The **utilib::ParameterSet::process_parameters** method is used to with command-line arguments. This method supports the use of the GNU style uses parameter keywords preceded by two hyphens rather than keyword letters. This style is extensible to contexts in which there are too many parameters to use single-letter parameters. This style is easier to read than the alphabet soup of older styles, and it can be combined with single-letter parameters (for commonly used parameters). A parameter argument (if any) can be separated by either whitespace or a single = (equal sign) character:

```
program --param1 paramval --param2=paramval
```

If a boolean parameter is specified without an argument, the argument is assumed to be **true**.

Note that the **utilib::ParameterSet::process_parameters** method requires three arguments. The first two are the standard **argc** and **argv** values provided in **main**. The third parameter is the number of required arguments that follow the optional parameters. Note that if this value is zero and optional arguments follow the command-line parameters, then the **utilib::ParameterSet** will assume that arguments following the last parameter are the value of that last parameter. For example

```
program --param1 <val>
```

in this case **utilib::ParameterSet** assumes that <val> is the value of parameter *param1*, even though parameter *param1* might be a boolean and the user intended <val> to be a regular argument to the program.

Several parameters are specified by default in the **utilib::ParameterSet** class. One of these is the *param-file* parameter, which can be used to specify a file that is read for parameter values. For example, the command

```
obj.set_parameter("param-file","foobar");
```

will trigger the **utilib::ParameterSet** class to open the file "foobar" and read it. The format of such an input file must be of the form: <parameter> <value>. Note that all or part of a line may be commented out using the "#" character. Additionally, if a parameter file contains a *param-file* parameter line, then that initiates the recursive opening of the specified parameter file.

5.1 Using ParameterList Objects

The use of a **utilib::ParameterList** object is motivated by the common scenario in which you wish to use a single file or set of command-line parameters to initialize the parameters in a set of classes. Rather than have each class process these data sources independently, the **utilib::ParameterList** class can be used to process all parameters at once and then set the values in each class. For example, if **Class1**, **Class2** and **Class3** are subclasses of **utilib::ParameterSet**, then we can do the following:

```
ParameterList plist;
plist.process_parameters(argc,argv,1);

Class1 c1;
c1.set_parameters(plist);
Class2 c2;
c2.set_parameters(plist);
Class3 c3;
c3.set_parameters(plist);
```

Note that **utilib::ParameterList** supports command-line processing and file IO exactly like the **utilib::ParameterSet** class. This example assumes that the parameters in each of these classes are independent, so when you set parameters they are removed from the parameter list in **plist**. If classes share parameter names that are defined in a consistent manner (e.g. a **debug** parameter), then you can keep parameters in the list by passing a flag "false" as the second argument to **utilib::ParameterSet::set_parameters**.

The **utilib::ParameterList** class also supports mechanisms that can be used to validate parameters that will later be used to initialize various class instances. This requires *registering* the parameters of all classes that will be used later:

```
ParameterList plist;
plist.register<Class1>();
plist.register<Class2>();
plist.register<Class3>();
plist.process_parameters(argc,argv,1);
```

Registering parameters before a **utilib::ParameterList** is initialized allows the **utilib::ParameterList** to verify that only parameters registered for use were provided in the command-line or input file. Note that this registration involves the construction of the specified object, so this can only be performed with classes that provide null-constructors.

The following example illustrates a typical setup to process command-line parameters after the start of main:

```
int main(int argc, char* argv[])
{
try {
  int debug=0;
  ParameterSet global_parameters;
  global_parameters.create_parameter("debug",debug,"<int>","0",
              "A global debugging parameter");

  ParameterList plist;
```

```
    plist.register(global_parameters);             // Register an existing
                                                   //   ParameterSet object
    plist.register<Class1>();                      // Register a class type that
                                                   //   will be used later
    plist.process_parameters(argc,argv,1);         // Process the command-line
    global_parameters.set_parameters(plist,false); // Set the values of the
                                                   //   global params, keeping
                                                   //   params in plist

                                                   // The "help" parameter
                                                   //   has been set to "true"
    if (global_parameters.get_parameter<bool>("help")) {
                                                   // Dump all of the registered
                                                   //   parameters to "cout"
       global_parameters.write_registered_parameters(cout);
       return -1;                                  // ... and exit.
       }

    // OTHER CODE HERE
    }
  catch (std::runtime_error& err) {
    cerr << "We caught an exception: " << err.what() << endl;
    }
return 0;
}
```

5.2 Parameter Validation

utilib::Parameter object includes data method **utilib::Parameter::validator** that can be used to validate that the value provided when a parameter is set is an appropriate value for that parameter. For example, we might wish to ensure that a string is not empty, or that a double is non-negative. When a parameter is created, an optional parameter can be specified that is an instance of an STL unary function, std::unary_function<Type,bool>, where *Type* is the type of the parameter data. A variety of such functions are provided for numerical parameters:

- **utilib::ParameterLowerBound** : enforces a lower bound on the parameter
- **utilib::ParameterUpperBound** : enforces an upper
- **utilib::ParameterBounds** : enforces upper and lower bounds on the parameter
- **utilib::ParameterNonnegative** : forces the parameter to be non-negative
- **utilib::ParameterPositive** : forces the parameter to be positive

For example, we can force a debugging parameter to be non-negative as follows

```
  create_parameter("debug",debug,"<int>","0",
        "A debugging parameter",
        utilib::ParameterNonnegative<int>());
```

6 Parallel Computing Utilities

Developing parallel software is notoriously difficult, particularly because of inherent difficulties associated with the debugging of parallel software. Parallel software coordinates threads of execution across multiple physical processors. Thus parallel software often exhibits programming errors related to timing and synchronization that are not seen in serial codes.

The remainder of this section describes UTILIB capabilities for developing and debugging parallel software.

6.1 MPI Utilities

MPI is a widely used standard for performing parallel communication on distributed-memory parallel computers. The UTILIB **mpi_utilib.h** header provides dummy definitions for `MPI_Request` and `MPI_Status`, which can be used to simplify builds when MPI is not available. Additionally, this header defines the **mpi_datatype()** functions, which return the MPI data type value for a data.

The **utilib::uMPI** class provides an object-oriented encapsulation of the MPI functions. In particular, this class manages global data for information like the number of processors, the rank of the current processor, etc. Further **utilib::uMPI** manages MPE log messages, and it automatically checks for error return codes for the MPI functions. Finally, **utilib::uMPI** identifies whether any of the current processors can do I/O. The following **utilib::uMPI** data elements are commonly used:

- comm - The MPI communicator.
- rank - The rank of the current process in this communicator.
- size - The size of the current communicator.
- ioProc - The rank of an I/O processor in this communicator.
- iDoIO - A true/false (1/0) value that indicates whether the current processor can perform I/O.

6.2 MPE Utilities

The **utilib::logEvent** class enables a code to generate parallel event log files that can be viewed with viewers like `upshot` and `jumpshot`. These viewers in turn allow the time sequence of events, thread activations, and messages to be visually inspected, an aid to debugging and development. Event logging uses the MPE extension library distributed with MPICH, and MPE is supposed to work with other flavors of MPI.

Event logging is compiled if UTILIB is configured to use MPI, MPE and validation. Otherwise, the event logging macros all compile into no-ops. Even if event logging is compiled, you must set the command line parameter "eventLog" to a non-zero value to obtain a log. For example, the PICO MILP solver is setup to take values between 0 and 4, where 0 means no log and 1 through 4 produce successively more detailed logs:

- Level 1: worker, incumbentCast, hub thread activations; problem bounding and new incumbent signaling states
- Level 2: level 1 + new incumbent messages, incumbent heuristic thread activations, pruning states
- Level 3: level 2 + status print events, and workerAux, spReceive, and spServe thread activations
- Level 4: level 3 + worker->hub messages

By default, a text file called "event.alog" is created, which can be viewed with the `upshot` viewer. If the environment variable MPE_LOG_FORMAT is set to CLOG, then the output is a binary file called "event.clog" which can be viewed with `jumpshot`. This mechanism is dictated by MPE, and our experience recommends the use of `jumpshot`.

The **IF_LOGGING_COMPILED** macro is used to add logging mechanism. For example, if

```
IF_LOGGING_COMPILED(int myLogState;)
```

is added into a class, then the variable `myLogState` is used to define a logging state. This value needs to be initialized before being used. For example, the following could appear in a constructor:

```
#ifdef EVENT_LOGGING_PRESENT
  myLogState = logEvent::defineState("name of state","corresponding X color");
#endif
```

Events are logged with the **LOG_EVENT** macro, which uses the syntax:

```
LOG_EVENT(level,what,myLogState)
```

Here, "level" is the minimum logging level to log the event. "what" is "start" to make the start of a state, "end" for the end, and "point" for start and end in quick succession (for logging "point events").

6.3 Parallel Communication with Packed Buffers

MPI provides a generic utility for packing data types into a buffer that can be reliably communicated between machines with different data representations. The **utilib::PackBuffer** and **utilib::UnPackBuffer** classes provide a convenient mechanism for creating these packed buffers, and for unpacking data that has been received from another process. In particular, these buffers leverage the C++ stream operators to allow for a simple syntax for packing and unpacking.

For example, here's a simple example of how a **utilib::PackBuffer** can be created

```
int i = -1;
char j[5] = "abcde";
utilib::PackBuffer pbuff;
pbuff << i;
pbuff << j;
```

The `buf()` returns a pointer to a buffer that contains the packed data, which is `size()` bytes long. A **utilib::PackBuffer** object can be reused by calling the `reset()` method.

The **utilib::UnPackBuffer** class has a similar usage. For example:

```
int i;
char j[5];
utilib::UnPackBuffer upbuff(buffer_data, buffer_len);
upbuff >> i;
upbuff >> j;
```

Note that the `setup()` method can also be used to initialize a **utilib::UnPackBuffer** object, so this class can be initialized after it is constructed.

Many UTILIB classes are instrumented to enable packing and unpacking with the stream operator. Further, the **utilib::PackObject** class can be used as a base class to define these stream operators. The derived class needs only redefine the `read()` and `write()` methods to make these stream operators functional.

Note that enumerated types need to be explicitly coerced to and from integer types in order to be used with the PackBuffer and UnPackBuffer classes. Consequently, it is convenient to define stream operators for enumeration types. The **ENUM_STREAMS** macro provides a generic mechanism for setting up these streams for an enumeration type.

For example, the COLIN library uses this macro to define stream operators for an enumerated type that describes bound constraints:

```
enum bound_type_enum    {
                        no_bound=0,
                        hard_bound=1,
                        soft_bound=2,
                        periodic_bound=3
                        };

ENUM_STREAMS( bound_type_enum )
```

6.4 Managing Parallel Output with a Common I/O Trace

6.4.1 Overview

Using print statements to trace interesting events is perhaps the simplest strategy for debugging software. However, there are several caveats for using them for parallel debugging. First, this technique can significantly impact the relative computation rates of processes during a parallel computation. Printing and especially file I/O are often very slow when compared to other computation tasks. Adding printing changes the behavior of asynchronous parallel programs so the precise error condition that a developer is tracking can disappear.

A second caveat for print-based debugging is that the order in which information is presented to a screen may not reflect the true sequence of events. For example, printing I/O for one process may be delayed while a buffer fills, allowing other processes' I/O to be displayed out of order. Explicitly flushing buffers (e.g. using `flush()`, can help, but even then communication delays can affect output ordering.

Finally, it is difficult (and at best inconvenient) to simultaneously display I/O from multiple processes, especially for an execution using hundreds to thousands of processors. Operating systems typically interleave the output of multiple processes, which can quickly lead to unintelligible output. One solution to this problem is to stream each process' I/O to a different file. In C, this can be done using `fprintf()` statements with a different file descriptor for each process. More sophisticated solutions can be developed in C++ by exploiting the extensibility of stream operators. The **utilib::CommonIO** class provides new streams **ucout** and **ucerr**, which replace `cout` and `cerr` to control I/O in a flexible manner. The **utilib::CommonIO** class ensures that I/O streamed to **ucout** and **ucerr** is printed as a single block, and thus it is unlikely to be fragmented on a screen. Additionally, each line of the output can be tagged with a processor id and line number:

```
[2]-00002 Printing line 2 from process 2
[3]-00007 Printing line 7 from process 3
[3]-00008 Printing line 8 from process 3
[1]-00003 Printing line 3 from process 1
[1]-00004 Printing line 4 from process 1
[0]-00003 Printing line 3 from process 0
```

This facility makes it easy to extract and order output for each process.

6.4.2 Using The CommonIO Class

The **CommonIO.h** header file provides macro-based definitions for the symbols **ucout**, **ucerr** and **ucin**. By default, these symbols are mapped to the standard C++ I/O streams. I/O can be directed to/from other streams using the `set_streams`, `set_cout`, `set_cerr` and `set_cin` methods. **utilib::CommonIO** also allows users to mask these streams, which adds information about the processor ID and/or line number of the I/O. To begin masking, the user executes `CommonIO::begin()`, and similarly `CommonIO::end()` is called to end the masking of these streams. For example:

```
CommonIO::begin()
ucout << "This text is rerouted through CommonIO's streams" << endl;
CommonIO::end()
```

Note that calls to `CommonIO::begin()` and `CommonIO::end()` can be nested, enabling subroutines to use **utilib::CommonIO** without worrying whether the IO has already been redirected. However, note that the formatting options for **utilib::CommonIO** streams described below are NOT reset after a call to `CommonIO::end()`; the user is responsible for resetting the state of the **utilib::CommonIO** streams after their use.

Because a begin-end block can be nested in this fashion, calling `CommonIO::end()` does not necessarily turn off IO mapping; IO mapping could have been initialized from an enclosing begin-end block. The method `CommonIO::map_off()` can be called to explicitly turn off IO mapping regardless whether mapping has been initiated

by a previous call to `CommonIO::begin()`. `CommonIO::map_on()` must be used to restart IO mapping; a subsequent call to `CommonIO::begin()` will be masked by the `CommonIO::off()` method. Finally, note that an error is detected if a matching `CommonIO::end()` is not called for every `CommonIO::begin()` when the last **utilib::CommonIO** object is destroyed.

utilib::CommonIO provides routines that manage parallel debugging, prepending tagging information, and providing a global debugging flag. Tagging information is prepended after `begin_tagging()` is called. An optional argument specifies the value of the `numDigits` variable (by default `numDigits=0`). The format of the prepended IO is:

```
[r]-000ii
```

where 'r' is the rank of the current process and 'ii' is the index of the current IO; the index field has numDigits digits. Tagging is stopped when `end_tagging()` is called.

To facilitate tagging, the **ucout** and **ucerr** macros map to `strstream` buffers. When the stream is flushed by calling `CommonIO::flush()` or by using the IO manipulator `Flush`,

```
ucout << Flush
```

the IO is processed to insert the rank information after every end-of-line in the stream. Using this facility makes it difficult to support the `flush` method for streams, since '<stream>.flush()' gets mapped to a flush operation on a `strstream` object. This does NOT flush the <stream> object as with standard `ucout` and `ucerr` streams. Consequently, the `Flush` IO manipulator has been created to provide a convenient means of flushing **utilib::CommonIO** streams.

utilib::CommonIO can also be used to buffer IO generated by a process. If `CommonIO::begin_buffered()` is called, then the IO is mapped via `strstream` objects, but the IO is not flushed until `CommonIO::end()` is called. If a new **utilib::CommonIO** begin/end block is started within a buffered begin/end block, IO within this block will continue to be buffered.

6.4.3 Managing Debugging IO

utilib::CommonIO also supports mechanisms for controlling I/O using a 'debugging level'. Consequently, **utilib::CommonIO** is often used a base class for other UTILIB classes.

The **utilib::CommonIO** class data member `debug` defines the 'debugging level' of the IO. This value is referenced by the `verbosity()` method, which determines if the given verbosity level is 'high enough' to be printed.

The debugging level is principally used by the **DEBUGPR** and **OUTPUTPR** macros, and related macros (see **CommonIO.h**). These macros accept two arguments: a debugging level and an arbitrary set of commands. These commands are executed if the specified debugging level is greater than or equal to the `CommonIO` debugging level.

The `setIORank()` method can be used to limit debugging IO to a limited number of processors. By default, the `verbosity()` check allows all processors to perform IO. By calling `setIORank(x)`, future calls to `verbosity()` will return `false` if `x` is not the rank of the current processor, thereby turning off debugging IO.

7 Miscellaneous Classes and Utilities

UTILIB provides a variety of general utility classes:

- **utilib::AnyValue** and **utilib::AnyReference** - Classes that store any object by value and reference respectively. The **utilib::AnyValue** class was adapted from the `boost::any` class, in particular to include UTILIB exception management and stream operators.

- **utilib::CachedAllocator** - A class that redefines the `new` and `delete` methods for a class to cache the allocation and deallocation of objects.

- **utilib::ClassRef** - A data type that manages the reference counting for unspecified data elements. This is a rather non-standard form of memory referencing, in which the reference object knows about all of the objects for which it is sharing memory. This is not a scalable form of reference management. However, it facilitates the use of reference sharing on an as-needed basis. Further, it facilitates the fast access of data in the main objects.

- **utilib::Ereal** - Defines an extension of 'real' data types (e.g. double, float, long double) that can 'assume' all 'real' values, as well as negative infinity and positive infinity. This is not meant to replace the use of IEEE arithmetic, but instead provide convenient container for managing data that may be infinite.

- **utilib::GenericKey** - A generic key object for use with UTILIB abstract data types (e.g. heaps, hash tables, etc).

- **utilib::PersistentPointer** - A class that looks like a pointer, but does not delete the underlying pointer's memory when this class is deleted!

- **utilib::RefCount** - An object used to maintain reference counts for shared data.

- **utilib::SmartPtr** - Class that manages pointer deletion and allows for sharing of pointers with reference counters

- Tuple# - The classes **utilib::Tuple1** ... **utilib::Tuple7** define n-tuples, which are fixed length lists.

- **utilib::ValuedContainer** - A container class that contains a value that can be extracted. For example, this is useful to sort objects that have a value with auxiliary information.

Additionally, the following UTILIB headers define miscellaneous utilities and support functions, including functions that are system-specific:

- **_generic.h** - Commonly used macros for 'standard' values (e.g. TRUE/FALSE)

- **comments.h** - Includes stream operators for processing comment lines and

- **memdebug.h** - Some experimental macros that can be used to debug memory allocation issues.

- **nicePrint.h** - Utilities for formatting output.

- **seconds.h** - Various utilities to query timing routines in a portable manner, including both system and wall-clock time.

- **std_headers.h** - Includes C and C++ headers to facilitate portability (especially for compilers that are not ANSI C++ compliant). white space.

- **stl_auxillary.h** - Defines functions and operators that can be applied to the std::vector class. In particular, this defines stream operators for this class.

- **string_ops.h** - Defines functions for manipulating **utilib::CharString** objects.

- **traits.h** - Defines a macro for setting global traits.

8 Managing Exceptions

A pervasive challenge for software development is the effective management of runtime error conditions (e.g. when an attempt to open a file fails). One effective strategy for debugging unexpected failures is to generate an `abort()` when an error is detected. This generates a core file with debugging information, including the call-stack at the point of failure.

Runtime error conditions are naturally managed in C++ code with the C++ exception construct. Exceptions allow for graceful, automatic management of erroneous states that require the code to step out of a given context back to a previous context. However, this general mechanism makes it difficult to debug the cause of an exception. Different compilers have different semantics for how exceptions are managed, and how to 'catch' an exception when using an interactive debugger. Hence, it is difficult to debug exception events in a generic manner.

The **exception_mngr.h** header defines macros that can be used to wrap exceptions in a generic fashion, trap exception events, and to manage what happens during an exception event. The **EXCEPTION_TEST** and **EXCEPTION_MNGR** macros can be used to encapsulate exception events for this utility. These macros should be called within source code like a function, except that a semicolon should not be added after the macro. For example, suppose that the exception `std::out_of_range` is thrown if `n > 100` in the file `my_source_file.cpp`. To use the macro, the source code would contain (at line 225 for instance):

```
EXCEPTION_TEST( n>100, std::out_of_range , "Error, n = " << n << is bad" )
```

When the program runs and with `n = 125 > 100` for instance, the `std::out_of_range` exception would be thrown with the error message:

```
/home/bob/project/src/my_source_file.cpp:225: n > 100: Error, n = 125 is bad
```

Similarly, the **EXCEPTION_MNGR** macro can be used

```
if (n>100)
   EXCEPTION_MNGR( std::out_of_range , "Error, n = " << n << is bad" )
```

which achieves the same result with an explicit conditional.

These macros call the **utilib::exception_mngr::handle_exception** function, which manages I/O and does one of the following:

- throws the exception (this is the default behavior)
- calls `abort()`
- calls `MPI_Abort()` when MPI is being used, and then calls `exit()`

The exception management can be configured to use these different options using the **utilib::exception_mngr::set_mode()** function. For example, the following configures this utility to call `abort()`:

```
utilib::exception_mngr::set_mode(utilib::exception_mngr::Abort);
```

This generates a core file in the same manner as described above for non-exception code debugging.

Exceptions can also be debugged interactively by using and setting an 'exit function' that is called before the exception is processed. The default exit function is **exit_fn()**, and an alternative exit function can be specified with the

utilib::exception_mngr::set_exit_function function. An interactive debugger can break on the execution of **exit_fn()**, which leaves the user at a point where the code state that generated the exception event can be analyzed.

Finally, note that the **exception_mngr.h** header also defines the **STD_CATCH** macro, which performs a catch for all standard exception types. This provides a convenient mechanism for ensuring that all possible exceptions will be caught in a C++ `main()` function.

9 Acknowledgements

The genesis of the UTILIB library is in the BBUMS library developed by Bill Hart and Brian Bartell while graduate students at U.C. San Diego. Although Brian would probably not recognize any of the UTILIB software, the design of some of the most widely used software, like array classes, is due to him. The BBUMS library was subsequently reorganized and renamed the SGOPT library, which focuses on a methods for global optimization. UTILIB was the stdlib subdirectory in SGOPT, which was extracted from SGOPT when it became clear that several groups at Sandia would be interested in using the UTILIB components without the additional baggage of the optimizers in SGOPT.

I would like to thank Cindy Phillips, Jonathan Eckstein, Mario Alleva, and Mike Eldred for their input on this software. Each of them has identified numerous bugs, and refinements in the configuration process are largely due to the demands that their uses of UTILIB have made.

UTILIB integrates and extends several tools developed by other authors:

- Many of the C files in utilib/src/ranlib are taken from the RANLIB.C library of C routines for random number generation, which was developed by Barry W. Brown and James Lovato, Dept of Biomathematics at the University of Texas, Houston. The C++ **utilib::RNG** class is adapted from the GNU class developed by Dirk Grunwald.
- The utilib::AbstractSplay class was adapted from code developed by D. Sleator, which itself is adapted from simple top-down splay, at the bottom of 669 of Sleator and Tarjan [6].
- The **utilib::AnyValue** class was adapted from the boost::any class developed by Kevlin Henney.
- Many of UTILIB's hashing functions call the general-purpose hash function published by Bob Jenkins in DDJ.
- I thank Roscoe Bartlett for many helpful discussions and for sharing his EXCEPTION macro, which inspired the exception management tools in **exception_mngr.h**.

This work was supported in part by the Mathematics, Information and Computational Science program, U.S. Department of Energy, Office of Energy Research, as well as the Laboratory Directed Research Program at Sandia National Laboratories. This work was performed at Sandia National Laboratories. Sandia is a multiprogram laboratory operated by Sandia corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

This document was prepared using the Doxygen software documentation tool, developed by Dimitri van Heesch, copyright 1997-2006. We are grateful to David Gay for editorial comments on a draft of this document.

References

[1] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1996.

[2] M. Evans, N. Hastings, and B. Peacock. *Statistical Distributions, Second Ed.* John Wiley and Sons, Inc., New York, 1993.

[3] D. E. Knuth. *The Art of Computer Programming. Volume 2, Seminumerical Algorithms*. 2nd edition.

[4] Lewis and Denenberg. *Data Structures and Their Algorithms*. Harper Collins, 1991.

[5] S. K. Park and K. W. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, October 1988.

[6] Sleator and Tarjan. Self-adjusting binary search trees. *JACM*, 32(3):652–686, July 1985.

[7] M. Weiss and B. Cummins. *Data Structure and Algorithm Analysis*. 1992.

[8] D. Wood. *Data Structures, Algorithms, and Performance*. Addison-Wesley, 1993.